Database Security

Paul Mooney
Chief Software Architect, Microsoft MVP

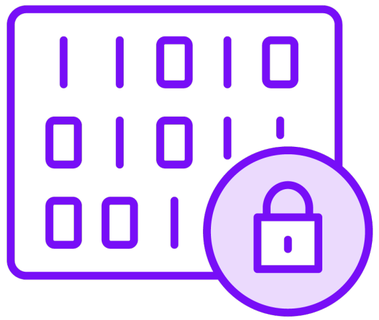
@daishisystems | www.insidethecpu.com

Overview

Coming Up

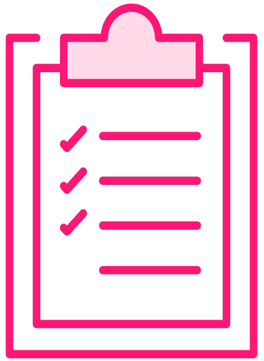
- Secure database configuration
- Unnecessary accounts, databases
- Input validation, output encoding
- Database authentication
- Secure code sharing on repositories
- Connection pool, lazy creation
- Access credentials, separate file
- Parameterized queries, prepared statements: SQL injection defense
- Stored procedures: security, access management, optimization

Best Practices

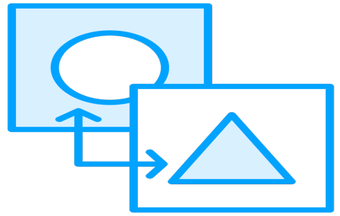


Secure installation
Root passwords
External access

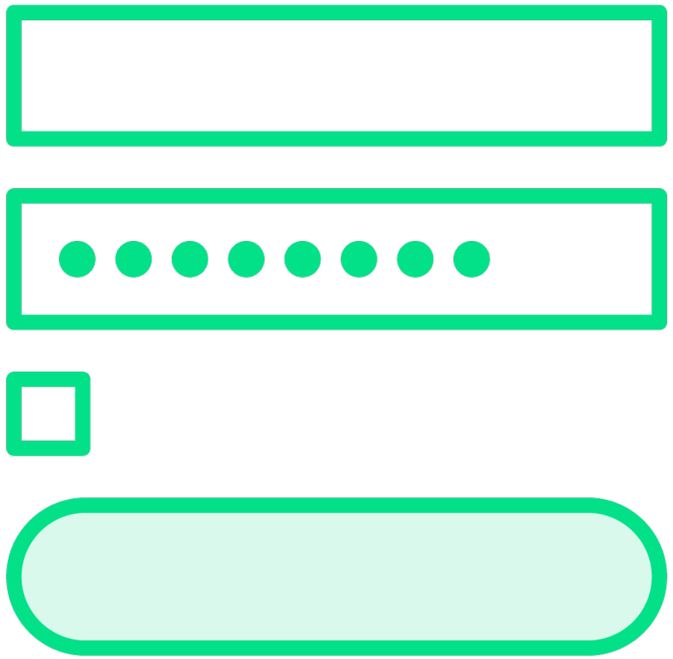
Anonymous-user accounts
Test databases
Unnecessary components



Required features
Unnecessary default accounts
Input, output



Not limited
Any language
All databases



Database Authentication

- Use read-only user for web app
- Minimize breaches, grant minimal privileges
- Choose strong password for database
- Use password manager for unique passwords
- Change/remove default admin passwords
- Beware of default "root" accounts
- Default accounts pose security risks
- Avoid including credentials in code
- Remove sensitive info before sharing

Database Connections

- Go's sql.Open returns *DB object
- Lazy connection creation for operations
- Use context-aware variants for operations
- Go's context triggers actions, manages resources
- Store auth details in separate file
- Avoid public directories for config file

Database Configuration File

```
<connectionDB>
  <serverDB>localhost</serverDB>
  <userDB>f00</userDB>
  <passDB>f00?bar#ItsP0ssible</passDB>
</connectionDB>
```

Reading the Configuration File

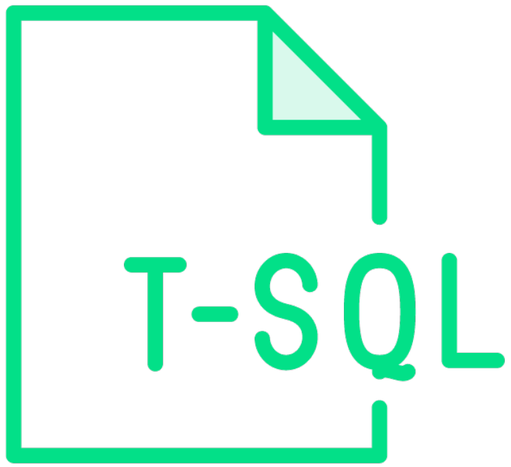
```
configFile, _ := os.Open("../private/configDB.xml")
```

Establishing a Database Connection

```
db, _ := sql.Open(serverDB, userDB, passDB)
```

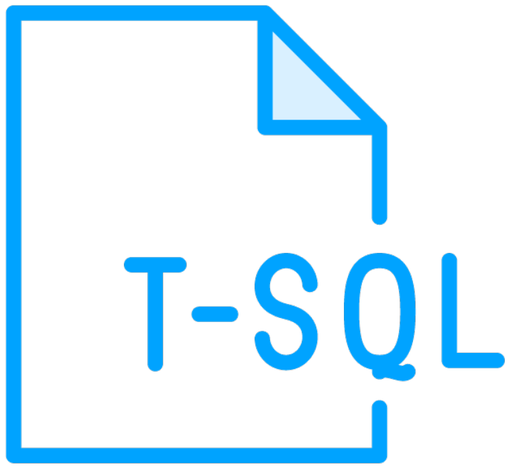
Best Practices

- Encrypt file for added security
- Use distinct credentials for different trust levels
- Separate credentials for regular, read-only, guest, and admin users



Parameterized Queries

- Secure approach to database interaction
- Prepared statements enhance security
- They defend against SQL injections
- Treat user-supplied data as data, not executable code
- Prepared statements may impact performance
- Explore alternative approaches when necessary
- Go's approach to prepared statements
- Statements are prepared on the DB object
- Associated with the appropriate connection from the pool

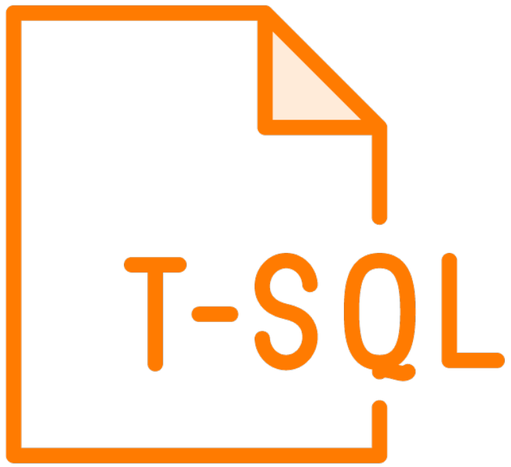


Prepared Statements Flow

- Prepare Stmt on a connection from pool
- Stmt retains connection information
- Stmt execution on same connection
- If unavailable, it looks for another connection
- Flow may result in high-concurrency database usage
- Numerous prepared statements can be created

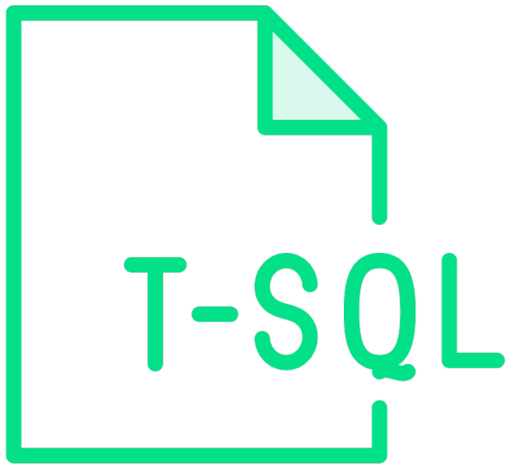
Prepared Statement with Parameterized Query

```
patientName := r.URL.Query().Get("name")
db.Exec("UPDATE patients SET name=? WHERE patientId=?", patientName, 233, 90)
```



When Not to Use Prepared Statements

- Database lacks support for prepared statements
- Statements not frequently reused, and security handled at another layer
- Performance optimizations prioritized over prepared statements



Stored Procedures

- Customized views control data access
- Prevent exposure of sensitive information
- Encapsulate complex logic within procedures
- Interface for specific procedure usage
- Differentiate information access
- Control tables and columns securely

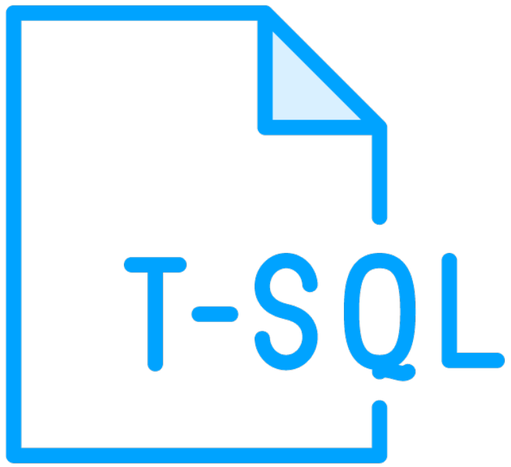
Stored Procedure Permissions

```
SELECT * FROM tblUsers WHERE userId = $user_input
```

Stored Procedure Permissions

```
CREATE PROCEDURE db.getName @userId int = NULL
AS
    SELECT name, lastname FROM tblUsers WHERE userId = @userId
GO

EXEC db.getName @userId = 14
```



Important Considerations

- Added layer for web app security
- DBAs gain permission control benefits
- Enhances server performance, reduces traffic

Summary

Database Security

- Best practices for securing databases in Go
- Input validation and output encoding for vulnerability protection
- Database authentication practices
- Database connections
- Parameterized queries and prepared statements for SQL injection prevention
- Benefits of stored procedures in controlling access to sensitive data